C10

Thu Jan 03 12:52:58 2008 Function Index Page 2

Thu Jan 03 12:52:58 2008 Function Index Page 1

Thu Jan 03 12:52:58 2008 Function Index Page 2

Thu Jan 03 12:52:58 2008 File Index Page 2

Thu Jan 03 12:52:58 2008 File Index Page 2

```
/*****************************************************************
                                                          ******
 **
 ** File Name:   RSLinitfin.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **              This module contains the Restore Service Library
                                                    functions to
 **              initialize and terminate the restore operation.
 **
 ** Table of Contents:
 ** ------------------
 **              RSTSL_Initialize
 **              RSTSL_Finish
 **
 **       Internal Functions:
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
 *****************************************************************
                                                          ******/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <dlfcn.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```



```
/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLstartup.h>


/*
 * #defines, structures, typedefs local to this source file
 */

static eerrno_ty init_plugins( restore_context *rcp );
static int validate_plugin( struct pluginData *piDataPtr );


/*
 * External declarations
 *
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;


/*
 * Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                                    These must be in the same order and position
 * as the PIFuncIndex values defined in RSLplugin.h.
 */

char *piFuncNames[PIFuncIndexLast+1] = {
        "RSTPI_Identify",
        "RSTPI_Initialize",
        "RSTPI_GetTopLevelObjects",
        "RSTPI_SetTopLevelObject",
        "RSTPI_GetNextLevelObjects",
        "RSTPI_ClearRestoreContext",
        "RSTPI_Submit",
        "RSTPI_GetTopLevelTemplates",
        "RSTPI_DoesAlternateExist",
        "RSTPI_MarkObject",
        "RSTPI_UnmarkObject",
        "RSTPI_IsObjectMarked",
        "RSTPI_IsObjectMarkable",
        "RSTPI_GetAllBackupTimes",
        "RSTPI_GetCurrentBackupTime",
        "RSTPI_SetBackupForTime",
        "RSTPI_SetPrevBackup",
        "RSTPI_SetNextBackup",
        "RSTPI_SetFirstBackup",
        "RSTPI_SetMostRecentBackup",
        "RSTPI_IsTherePrevBackup",
        "RSTPI_IsThereNextBackup",
        "RSTPI_IsTherePrevBackupForTime",
        "RSTPI_IsThereNextBackupForTime",
        "RSTPI_Finish",
        "RSTPI_StartRestore",
        "RSTPI_FindRestorableObjects",
        "RSTPI_GetFindResults",
        "RSTPI_GetNecessaryMedia"
};
```

```
127  /*****************************************************************
128   * RSTSL_Initialize:
129   *
130   *  This function takes care of all the initialization for a restore
131   *  session.  This must be called prior to any of the other functions
132   *  in the Restore API.
133   *
134   * Parameters:
135   *
136   *  userName (I) - The name of the user.
137   *
138   *****************************************************************/
140  eerrno_ty
141  RSTSL_Initialize( const char *userName )
142  {
143      eerrno_ty status = E_SUCCESS;
145      /*
146       * If we have not yet allocated space for a restore_context
147       * structure, do so now.  If we have already done so,
                                                        just clear it
148       * now.
149       */
151      if (NULL == rcp)
152      {
153          rcp = (struct restore_context *)malloc(sizeof(
                                                struct restore_context));
154          if (NULL == rcp)
155           {
156               rec_api_log_csm(SUB_CSM_NOMEM, NULL);
157               return(EP_RB_RECOVER_NOMEM);
158           }
159      }
160      memset(rcp, 0, sizeof(struct restore_context));
162      rcp->rc_human_uidname = esl_strdup( userName );
165      if (!rcp->rc_human_uidname) {
166        rec_api_log_csm(SUB_CSM_NOMEM, NULL);
167        return(EP_RB_RECOVER_NOMEM);
168      }
170      /*
171       * Set the appropriate field in the recovery context to indicate
172       * that this recover session is based on the Recover API.
173       * This flag is in place for historical reasons but is used by
174       * other parts of the Recover API library.
175       */
177      rcp->gui_mode = 1;
179      /*
180       * Initialize the logging mechanism.
181       */
183      if (status = rbrlog_begin(rcp, progname))
184      {
185          return(status);
186      }
188      /*
189       * Initialize the few "recover context" variables that we can at
190       * this early stage.
```



```
191      */
193      setup_proc(rcp);
195      /*
196       * The following call will:
197       *    -Initialize the saveset datatbase.
198       *    -Infer any information we can at this point.
199       */
201      if (status = startup(rcp))
202      {
203          return(status);
204      }
206      /* Do plugins setup: Find and initialize all valid restore plugin
                                                                  libs: */
208      status = init_plugins( rcp );
210      return( status );
211  }       /* End of RSTSL_Initialize() */
```

```
214     /*****************************************************************************
215      * RSTSL_Finish
216      *
217      *    Function Description:
218      *
219      *  This function terminates a restoral session,
                                                but not while a restore is in
220      *  progress.
                 It will be rejected if a restore is currently being executed.
221      *  This routine will clean up any local memory used in the session.
222      *
223      * Parameters:
224      *
225      *  none
226      *
227      */
228
229     eerrno_ty
230     RSTSL_Finish( void )
231 1   {
232 1       int mc_n;
233
234
235 1       eerrno_ty err = E_SUCCESS;
236
237 1       if (NULL == rcp)
238 2       {
239 2           return( E_SUCCESS );
240 1       }
241 1       RemoveSubmitFiles();
242 1       /*
243 1        * Call rbr_cleanup() which kills the aux proc(
                                                        s), unlocks the work
244 1        * item, then calls rbrlog_end(
                                                ) to enter the last logs and to close
245 1        * the log file.
246 1        */
247
248 1       rbr_cleanup(rcp);
249
250 1       /*
251 1        * Deallocate the memory of restore_context and the related
252 1        * structures.
253 1        */
254
255 1       if (NULL != rcp->rc_mcp)      /* Free the multicat structures */
256 2       {
257 2           mcat_destroy(rcp->rc_mcp);
258 1       }
259
260 1       /*
261 1        * Free the mark bit map space
262 1        */
263
264 1       for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
265 2       {
266 2           if (NULL != rcp->rc_marks[mc_n])
267 3           {
268 3               free(rcp->rc_marks[mc_n]);
269 2           }
270 2           rcp->rc_marks[mc_n] = NULL;
271 1       }
272
273 1       if (NULL != rcp->rc_nmarks_by_plane)
```



```
274 2   {
275 2       free(rcp->rc_nmarks_by_plane);
276 1   }
277
278 1   /*
279 1    * Free the configuration structures
280 1    */
281
282 1 #if 0
283 1   if (NULL != rcp->rc_cfgname)
284 2   {
285 2       free(rcp->rc_cfgname);
286 1   }
287 1 #endif
288
289 1   if (NULL != rcp->rc_config)
290 2   {
291 2       rbc_freeconfig(rcp->rc_config);
292 1   }
293
294 1   /*
295 1    * Free the DS_NONE structures array
296 1    * Note that even though rc_dsnones is the head of linked list
297 1    * of dsnone_info structures, the list is allocated via malloc
298 1    * as an array initially (ref. alloc_plane_arrays()), therefore
299 1    * we can do a free here.
300 1    */
301
302 1   if (NULL != rcp->rc_dsnones)
303 2   {
304 2       free(rcp->rc_dsnones);
305 1   }
306
307 1   /*
308 1    * Free the volume list structures.
309 1    */
310
311 1   if (NULL != rcp->ebvllist)
312 2   {
313 2       (void)ebvl_volidlist_destructor(
                                            rcp->ebvllist, EBVL_DESTROY_ALL);
314 1   }
315
316 1   /*
317 1    * Free the plugin related data
318 1    */
319
320 1   rcp->rc_backup_app = 0;
321 1   while (rcp->currentPIptr = rcp->piList)
322 2   {
323 2       rcp->rc_backup_app++;
324 2       rcp->appData = rcp->currentPIptr->appData;
325 2       /* allow plugin to clean up and close .so: */
326 2       if ( E_SUCCESS != (err =
327 2               rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
                                                            rcp) ) )
328 3       {
329 3       /* log error, continue */
330 3           rbe_user_error( err,
331 3                   "RSTPI_Finish failed for restore plug-in
                                                        library %s\n",
332 3                   ((struct pluginIDdata *)(
                                rcp-> currentPIptr-> idData))-> name );
333 2       }
334 2       dlclose( rcp-> currentPIptr-> libHdl );
335 2       rcp->piList = rcp->piList->next;
```

```
336 2           free (rcp->currentPIptr);
337 1       }

339 1       /*
340 1        * Free the various simple string buffers
341 1        */

343 1       if (NULL != rcp->rc_top_level_object_name)
344 2       {
345 2           free(rcp->rc_top_level_object_name);
346 1       }

348 1       if (NULL != rcp->rc_template_name)
349 2       {
350 2           free(rcp->rc_template_name);
351 1       }

353 1       if (NULL != rcp->rc_workitem_name)
354 2       {
355 2           free(rcp->rc_workitem_name);
356 1       }

358 1       if (NULL != rcp->rc_human_uidname)
359 2       {
360 2           free(rcp->rc_human_uidname);
361 1       }

363 1       if (NULL != rcp->rc_effective_uidname)
364 2       {
365 2           /* don't free, its internal: free(rcp->rc_effective_uidname);
                                                                        */
366 1       }

368 1       if (NULL != rcp->rc_client_rbuname)
369 2       {
370 2           free(rcp->rc_client_rbuname);
371 1       }

373 1       if (NULL != rcp->rc_client_hostname)
374 2       {
375 2           free(rcp->rc_client_hostname);
376 1       }

378 1       if (NULL != rcp->rc_client_scriptname)
379 2       {
380 2           /* don't free, its internal: free(rcp->rc_client_scriptname);
                                                                        */
381 1       }

383 1       if (NULL != rcp->rc_client_dirtop)
384 2       {
385 2           free(rcp->rc_client_dirtop);
386 1       }

388 1       if (NULL != rcp->rc_cmd_context)
389 2       {
390 2           /* don't free -- its internal/temp data: free(
                                                rcp->rc_cmd_context); */
391 1       }

393 1       if (NULL != rcp->rc_source_client_hostname)
394 2       {
395 2           free(rcp->rc_source_client_hostname);
396 1       }

398 1       if (NULL != rcp->rc_cpiogen_executable)
```



```
399 2       {
400 2           /* don't free, its internal: free(rcp->rc_cpiogen_executable);
                                                                        */
401 1       }

403 1       if (NULL != rcp->rc_plugin_wi_types)
404 2       {
405 2           free(rcp->rc_plugin_wi_types);
406 1       }

408 1       if (NULL != rcp->rc_pwd)
409 2       {
410 2           free(rcp->rc_pwd);
411 1       }

413 1       /*
414 1        * Finally, deallocate the restore_context itself
415 1        */

417 1       free(rcp);
418 1       rcp = NULL;



422 1       return( err );
423 }           /* RSTSL_Finish */
```

```
427  /*****************************************************************
428   * init_plugins
429   *
430   *    Function Description:
431   *
432   *  This function locates, opens, validates and initializes all restore
433   *  plug-in (shared) libraries.  They must be located in
434   *  /usr/epoch/EB/cure_plugin (eb_cure_plugin_dir).  All .so files in that
435   *  directory are opened and validates for version# and presence of all
436   *  mandatory functions.  The RSTPI_IDentify function is called for each
437   *  library to determine which optional features are supported, and that
438   *  the corresponding functions are present.  Finally, the RSTPI_Initialize
439   *  function is called for each valid library.
440   *
441   * Parameters:
442   *
443   * Inputs:
444   *       rcp     (I)     - Pointer to restore context
445   *
446   * Outputs:
447   *       none
448   *
449   * Returns:
450   *       E_SUCCESS or EP_RB_RECOVER_xxx
451   *
452   * Logic/pseudo code:
453   *
454   *       open plugin dir
455   *       while read_next_entry succeeds
456   *             verify .so file (else continue)
457   *             open shared library file (else continue)
458   *             on errors below:
459   *                   close shared library file
460   *                   continue
461   *             fetch all mandatory function addresses
462   *             call IDentify function
463   *             validate version number
464   *             fetch all indicated optional function addrs
465   *             call Initialize function
466   *             add workitem types to composite exclusion list
467   *             add to valid plugin list
468   *       close plugin dir
469   */

471  static eerrno_ty init_plugins( restore_context *rcp )
472  {
473          DIR             *dirp;
474          struct dirent   *direntp;
475          eerrno_ty       status = E_SUCCESS;
476          struct pluginData *piDataPtr = NULL;
477          struct pluginData *piListPtr = NULL;
478          int             val_result;
479          struct pluginIDdata *idDataPtr;
480          char            *tmp_types;
481          int             shlib_dirlen;
482          char            shlib_path [MAXPATHLEN];
```



```
484          /* open plugin directory or bust */
485          if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
486          {
487                  rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
488  #if 1
489                  return E_SUCCESS;
                                /* allow continuation w/o plugins */
490  #else
491                  return EP_RB_RECOVER_NO_PLUGINS;
                                                /* later do this */
492  #endif
493          }

495          strcpy( shlib_path, eb_cure_plugin_dir );
496          strcat( shlib_path, "/" );
497          shlib_dirlen = strlen (shlib_path);

499          /* loop thru entries in directory*/
500          while (NULL != (direntp = readdir( dirp )))
501          {
502                  if (NULL == piDataPtr)
503                  {       /* allocate next plugin data structure */
504                          if (NULL == (piDataPtr
505                                          = calloc( 1, sizeof(
                                                  struct pluginData) )))
506                          {
507                                  status = EP_RB_RECOVER_NOMEM;
508                                  break;
                                                /* fall thru to cleanup */
509                          }
510                  }

512                  if (NULL == strstr( direntp->d_name, ".so" ) )
513                          continue;       /* skip this guy */

515                  strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
516                  if (NULL == (piDataPtr->libHdl
517                                  = dlopen( shlib_path, RTLD_NOW )))
518                  {
519                          rbe_user_error( 0,
520                                  "Error opening restore plug-in library
                                                                  %s: %s\n",
521                                  direntp->d_name, dlerror() );
522                          continue;       /* skip this one */
523                  }

525          /* Fetch addresses of all mandatory functions and */
526          /* Do IDentify processing: call it, save options, validate */

528                  if (0 != (val_result = validate_plugin(
                                                          piDataPtr ) ))
529                  {
530                      if (val_result == -1 || val_result == -4)
531                      {
532                          rbe_user_error( 0,
533                  "Functions missing from restore plug-in library %s:
                                                                  %s\n",
534                                  direntp->d_name, dlerror() );
535                      }
536                      else if (val_result < 0)
537                      {
538                          rbe_user_error( 0,
539                              "Validation failed for restore plug-in
                                                          library %s\n",
540                                  direntp->d_name );
541                      }
```

```
542 3                       else
543 4                       {
544 4                           rbe_user_error( val_result,
545 4                   "RSTPI_IDentify failed for restore plug-in library
                                                                %s\n",
546 4                                       direntp->d_name );
547 3                       }

549 3                       dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
550 3                       piDataPtr->libHdl = NULL;
551 3                       continue;   /* on any error, skip this lib */
552 2                   }

554 2           /* let DC plug-in do its initialization */
555 2                   rcp->appData = NULL;
                                        /* enter plugin with clean appdata */
556 2                   status =
                       piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
557 2                   if (E_SUCCESS != status)
558 3                   {
559 3                           rbe_user_error( status,
560 3                   "RSTPI_Initialize failed for restore plug-in library
                                                                %s\n",
561 3                                       direntp->d_name );
562 3                           dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
563 3                           piDataPtr->libHdl = NULL;
564 3                           status = E_SUCCESS;
                                                /* this was't fatal */
565 3                           continue;
                                        /* on any error, skip this lib */
566 2                   }

568 2           /* save plugin's appData */
569 2                   piDataPtr->appData = rcp->appData;
570 2                   rcp->appData = NULL;

572 2           /* add piDataPtr to valid plugin list */
573 2                   if (NULL == piListPtr)
574 2                           rcp->piList= piDataPtr;
                                                /* first in list */
575 2                   else
576 2                           piListPtr->next = piDataPtr;
                                                /* link from prev */
577 2                   piListPtr = piDataPtr;
                                                /* new end of list */
578 2                   piDataPtr = NULL;

580 2           /* add workitem types to composite exclusion list */
581 2                   idDataPtr = (struct pluginIDdata *)piListPtr->idData;
582 2                   if (idDataPtr->num_types > 0)
583 3                   {
584 3                       tmp_types = calloc( 1, 1 + idDataPtr->num_types
585 3                                           + rcp->rc_num_plugin_wi_types
                                                                        );
586 4                       if (NULL == tmp_types) {
587 4                           status = EP_RB_RECOVER_NOMEM;
588 4                           break;
589 3                       }
590 3                       if (NULL != rcp->rc_plugin_wi_types)
591 4                       {
                          /* move old list to new buffer and free old list */
592 6                           memcpy( tmp_types,
593 6                                   rcp->rc_plugin_wi_types,
594 4                                   rcp->rc_num_plugin_wi_types );
```



```
595 4                           free( rcp->rc_plugin_wi_types );
596 3                       }
597 3                       memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
598 3                               idDataPtr->wi_types,
599 3                               idDataPtr->num_types );
600 3                       rcp->rc_num_plugin_wi_types +=
                                                    idDataPtr->num_types;
601 3                       tmp_types[rcp->rc_num_plugin_wi_types] = 0;
602 3                       rcp->rc_plugin_wi_types = tmp_types;
603 2                   }
604 1           }

606 1           (void)closedir( dirp );

608 1           /* free up leftovers: */
609 1           if (NULL != piDataPtr)
610 1                   free (piDataPtr);

612 1           if (E_SUCCESS != status)
613 2           { /* Free contents of plugin list: */
614 2                   while (NULL != (piDataPtr = piListPtr))
615 3                   { /* allow plugin to clean up and close .so: */
616 3                           rcp->appData = piDataPtr->appData;
617 3                           piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                                    rcp );
618 3                           dlclose( piDataPtr->libHdl );
619 3                           piListPtr = piDataPtr->next;
620 3                           free (piDataPtr);
621 2                   }
622 1           }

624 1           return status;
625   }
```

```
626         /* init_plugins */

628 /*****************************************************************
629  * validate_plugin
630  *
631  *    Function Description:
632  *
633  *  This function retrieves the addresses of the mandatory plugin
                                                              functions
634  *  and stores them in the function pointer array.
                                                 If any function is missing
635  *  it returns -1.
                It then calls the identify function and verifies wthe plugin
636  *  version,
                and finds its optional functions. Specific error values are
637  *  returned on version mismatch and missing optional functions.
638  *
639  * Parameters:
640  *
641  * Inputs:
642  *       piDataPtr (
                  I)    - Pointer to plugin data structure with libHdl set
643  *
644  * Outputs:
645  *       piFuncArray in piDataPtr is loaded with pointers to plugin
                                                              functions
646  *
647  * Returns:
648  *       0 on success
649  *       -1 on any missing required functions
650  *       -2 if version validation fails OR identify returns junk
651  *       -3 if workitem type validation fails
652  *       -4 on any missing optional functions indicated by options
                                                                  flags
653  *       +n (
     EB_RB_RECOVER_xxx) for error codes returned from Identify function
654  *
655  *****************************************************************
                                                              ****/

657 static int validate_plugin( struct pluginData *piDataPtr )
658 1 {
659 1         int             index;
660 1         eerrno_ty       status;
661 1         struct pluginIDdata *idDataPtr;

663 1         for( index = 0; index <= PIFuncIndexLastBasic; index++ )
664 2         {
665 2                 if (NULL == (piDataPtr->piFuncArray[index]
666 2                         = (piFuncPtr) dlsym( piDataPtr->libHdl,
667 2                                                 piFuncNames[index]
                                                                    ))
668 2                 )
669 2                         return -1;
670 1         }

672 1         /* call identify and validate: */
673 1         status = piDataPtr->piFuncArray[PIFuncIndexIdentify](
674 1
                                                    &piDataPtr->idData );
675 1         if (status != E_SUCCESS)
676 1                 return status;
677 1         if (NULL == (idDataPtr =
                          struct pluginIDdata *)piDataPtr->idData) )
678 1                 return -2;
```



```
680 1         if (idDataPtr->version != RSTPI_VERSION)
681 2         { /* only version 1 supported so far */
682 2                 piDataPtr->idData = NULL;
683 2                 return -2;
684 1         }

686 1         if (idDataPtr->num_types && !idDataPtr->wi_types)
687 2         {       /* count cant be positive with null pointer */
688 2                 piDataPtr->idData = NULL;
689 2                 return -3;
690 1         }

692 1         /* if startRestore option set, get its addr or bust */
693 1         if ( ( (RSTPI_OPTION_SPECIAL_START
694 1                 == (idDataPtr->options & RSTPI_OPTION_MASK_START))
695 1              && (NULL == (
                                piDataPtr->piFuncArray[PIFuncIndexStartRestore]
696 1                                 = (piFuncPtr) dlsym( piDataPtr->libHdl,
697 1
                                                piFuncNames[PIFuncIndexStartRestore] )))
698 1                 )
699 1         /* OR if special find option set, get its addr or bust */
700 1           || ( (RSTPI_OPTION_SPECIAL_FIND
701 1                 == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
702 1              && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFind]
703 1                                 = (piFuncPtr) dlsym( piDataPtr->libHdl,
704 1
                                                piFuncNames[PIFuncIndexFind] )))
705 1                   || (NULL == (
                                piDataPtr->piFuncArray[PIFuncIndexFindResults]
706 1                                 = (piFuncPtr) dlsym( piDataPtr->libHdl,
707 1
                                                piFuncNames[PIFuncIndexFindResults])
708 1                   ) ) )         )
709 1         /* OR if special getMedia option set, get its addr or bust */
710 1           || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
711 1                 == (
                                idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
712 1              && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
713 1                                 = (piFuncPtr) dlsym( piDataPtr->libHdl,
714 1
                                                piFuncNames[PIFuncIndexGetMedia] )))
715 1         )     )
716 2         {
717 2                 piDataPtr->idData = NULL;
718 2                 return -4;
719 1         }

721 1         return 0;
722     }
```

```
723  /*      validate_plugin */
```

```
#define _POSIX_SOURCE 1

#include <restore/restoretree.h>
#include <restore/RSLplugin.h>
#include <ebconfig/rbconfig.h>


//
```



```
extern "C" void dump_unix_time(RSTRPC_time_ty time)
{
    DateTime t(time);
    if(time)
      {
        cout << "Time: " << t << endl ;
      }
    else
      {
        cout << "Time: Zero value\n";
      }
}
```

```
23
24  1 extern "C" void dump_time_list(RSTRPC_time_list *list, ostream &out)
25  1 {
26  1   DateTime *t;
    1   out << "Dump of time list\n";
28  1   for(RSTRPC_time_list *listelem=list;listelem;
                                          listelem=listelem->next)
29  2     {
30  2       t=new DateTime(listelem->time);
31  2       out << " Time: " << *t << endl;
32  2       free (t);
33  1     }
    1   out << "End of time list\n";
35  1
37    }
```



```
39
40  1 extern "C" void dump_tlo_list(RSTRPC_tlo_list *list,ostream &out)
41  1 {
    1   out << "Dump of RSTRPC_tlo_list\n";
42  1   for(;list;list=list->next)
43  2     {
44  2       RSTRPC_top_level_obj *tobj=list->tlo;
45  2       RSTRPC_restorable_obj_root *ro=&(tobj->root);
46  2       out << "TLO " << endl;
47  2       out << "       level:" << ro->objLevel << " Backup App:"<<
                                                    ro->backupApp<<endl;
48  2       if(ro->objName)
49  3         {
50  3           out << "       name :" << ro->objName <<endl;
51  2         }
52  2       if(ro->objTypeString)
53  3         {
54  3           out << "       types:" << ro->objTypeString << endl;
55  2         }
56  1     }
57  1   out << "End of dump RSTRPC_tlo_list\n";

59  1   return;
60    }
```

```
62  //
```



```
63
64  1  extern "C" void dump_uro_list(RSTRPC_uro_list *list,ostream &out)
65  1  {
66  1    out << "Dump of RSTRPC_uro_list\n";
67  2    for(;list;list=list->next)
68  2      {
69  1        out << "We have a node\n";
70  1      }
71    out << "End of dump RSTRPC_uro_list\n";
  }
```

```
73  //
```



```
74  eerrno_ty RSTPI_Initialize(restore_context *context)
75  1 {

77  1   // See if an initial sleep is needed
78  1   /*  char *sl=getenv("RSTPI_INITIALIZE_SLEEP");
79  1   if(!sl)
80  1     {
81  1       int st=atoi(sl);
82  1       rbe_log_stats(0,"RSTPI_Initialize, sleep of %d seconds",st);
83  1       sleep(st);
84  1     }
85  1   else
86  1     {
87  1       rbe_log_stats(0,"RSTPI_Initialize, no delay on startup");
88  1     }
89  1     */
90  1   // Allocate restore context data
91  1   context->appData=(void *)malloc(sizeof(struct restoreContextData));

93  1   if(!context->appData)
94  2     {
95  2       rbe_log_stats(0,"plugin.cc - malloc failure");
96  2       return EP_RB_RECOVER_MALLOC_FAILURE;
97  1     }

99  1   struct restoreContextData *rcd=
100 1     (struct restoreContextData *)(context->appData);

102 1   rcd->currentWisetListNode=NULL;
103 1   rcd->currentWisetNode=NULL;
104 1   rcd->currentBackupNode=NULL;

106 1   return E_SUCCESS;
107 }
```

```
108 //
```



```
110 //
```

```
111   /*****************************************************************
112    * Submit
113    *
114    * This function creates a submit object from the currently marked
115    * restorable objects.                    The ID of the created submit object is passed to
116    * EDMRST_Start to begin execution of the restore.
117    *
118    * Parameters:
119    *
120    * context        (I) - Pointer to the restore context
121    * hostName       (I) - host to restore to (only if inPlace == False)
122    * policy         (I) - The overwrite policy to use
123    * inPlace        (                                                    I) - Flag if the restoral is to be in original locations
124    * directory      (I) - directory to restore to (                                                    only if inPlace == False)
125    * transport      (                                                    I) - Indicator of transport the restoral is to be over (SCSI
126    *                                                                     or network)
127    * submitObjIDptr(                                                    IO) - ID of the submit user object created to describe
128    *                                                                     the restore
129    * progressCB     (                                                   I) - pointer to callback function to report progress and
130    *                                                                     test for cancellation
131    *
132    *  Note:
133    *         The Progress Callback is currently not used.  If the need
134    *         for this developes, the routines which use this argument
135    *         (mark, unmark, submit) must be enhanced.
136    *
137    *****************************************************************/
138   errno_ty RSTPI_Submit( restore_context          *context,
139                          const char               *hostName,
140                          const OverwritePolicy    policy,
141                          const boolean_ty         inPlace,
142                          const char               *directory,
143                          const RestoreTransport   transport,
144                          int                      *submitObjIDptr,
145                          RSTPI_SubmitProgressProc progressCB )
146 1 {


149 1   // Get the restore context so we can find the app data
150 1   //
151 1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);

153 1   if(!rcd)
154 2     {
155 2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
156 1     }

158 1   //
159 1   // Get the current backup node from the top level object
160 1   //
161 1   if(!rcd->currentBackupNode)
162 2     {
163 2       return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
164 1     }



167 1   BackupNode *cbu=rcd->currentBackupNode;

169 1   bool scode=cbu->fillSubmitObject(context,
```



```
170 1                                       hostName,
171 1                                       policy,
172 1                                       inPlace,
173 1                                       directory,
174 1                                       transport,
175 1                                       submitObjIDptr,
176 1                                       progressCB);
177 1   if(!scode)
178 2     {
179 2       rbe_log_stats(0,"RSTPI_Submit - error from fillSubmitObject");
180 2       return EP_RB_RECOVER_FILLSUBMIT_ERROR;
181 1     }


184 1   return E_SUCCESS;

186   }
```

```
190 /*****************************************************************************
191  * Get Top Level Objects
192  *
193  *  This function is called to retrieve the configurable backup
                                                          objects (work
194  *  items for network backups and work item sets for Symmetrix Connect
                                                          backups,
195  *  which are restorable for the given client.
196  *
197  *  It is a GOAL of this routine to return all objects ever backed
198  *  up successfully.  For network backups, though,
                                                  it only looks in the config
199  *  file for 'top level objects' of the given client.
200  *
201  *  While the restore API will be called repeatedly to retrieve a
                                                          maximum
202  *  number of items on each call,
                                  this plug-in call must retrive the whole
203  *  set of applicable backup objects.
                                       The generic restore service library
204  *  will manage the composite list of top level objects from all
                                                          backup apps.
205  *
206  * Parameters:
207  *    context        (I)  - Pointer to the restore context
208  *    sourceHost     (
               I) - the name of the host whose backups are being restored
209  *    topLevObjs     (O)  - ptr to linked list of Top Level Objects
210  *    numberEntries (
                    O)  - the real number of objects returned in the list
211  *
212  * Returns:
213  *        E_SUCCESS            on success
214  *        EP_RB_RECOVER_xxx  on error
215  *
216  *****************************************************************************/

218 eerrno_ty RSTPI_GetTopLevelObjects(restore_context *context,
219                                    const char * sourceHost,
220                                    struct RSTRPC_tlo_list
                                                          **topLevelObjs,
221                                    short *numberEntries)
222 1 {
223 1   //
224 1   // Now look through current backups and put on list if in
225 1   // Range
226 1   //
227 1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                          context->appData);


229 1   if(!rcd)
230 2     {
231 2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
232 1     }



235 1   rcd->currentWisetListNode=new WiSetListNode((char *)sourceHost);



238 1   //
239 1   // Populate this node only
240 1   //
241 1   if(!rcd->currentWisetListNode->populate(
```



```
                                                  context,POPULATE_NODE,sourceHost))
242 2     {
243 2       return EP_RB_RECOVER_WISETLIST_NODE_POPULATE_ERR;
244 1     }

246 1   *numberEntries=0;
247 1   *topLevelObjs=NULL;

249 1   RSTRPC_tlo_list *previous=NULL;


252 1   for(RestoreNode *rn    =rcd->currentWisetListNode->getFirstChild();
253 1       rn;
254 1       rn=rcd->currentWisetListNode->getNextChild())
255 2     {
256 2       (*numberEntries)++;
257 2       //
258 2       RSTRPC_tlo_list *new_list_elem=(RSTRPC_tlo_list *)malloc(sizeof(
                                                  struct RSTRPC_tlo_list));

259 2       if(!new_list_elem)
260 3         {
261 3           rbe_log_stats(0,"plugin.cc - malloc failure");
262 3           return EP_RB_RECOVER_MALLOC_FAILURE;
263 2         }
264 2       RSTRPC_top_level_obj *new_top=(RSTRPC_top_level_obj *)malloc(
                                          sizeof(struct RSTRPC_top_level_obj));

265 2       if(!new_top)
266 3          {
267 3            rbe_log_stats(0,"plugin.cc - malloc failure");
268 3            return EP_RB_RECOVER_MALLOC_FAILURE;
269 2          }
270 2       memset(new_top, 0, sizeof(struct RSTRPC_top_level_obj));



273 2       /*
274 2        * We need to check to see if a network is restorable for this
275 2        * top level object.  Since the client machines are the same for
276 2        * all backups of the same workitem (duh!) we only need to check
277 2        * one. Although this should always work, we will behave in a
278 2        * failsave manner by
279 2        *
280 2        * (a) Macking sure the child is non-null
281 2        * (b) Marking network as "not possible" when we find that case
282 2        *     (so we consider network restore possible, and an allowed
283 2        *      option, if we don't find a client in the ddtab for the
284 2        *      first backup of the work item)
285 2        */
286 2       BackupNode *bn=(BackupNode *)rn->getFirstChild();
287 2       // C++ standard evaluates left to right, so this is null-safe
288 2       if(bn && !bn->getNetworkRestorePossible())
289 3         {
290 3           new_top->flags |= TLO_BITFLAG_NETWORK_RESTORE_NOT_POSSIBLE;
291 2         }

293 2       char ** temp_ptr;
294 2       temp_ptr=(char **)malloc(sizeof(char *));
295 2       if(!temp_ptr)
296 3          {
297 3            rbe_log_stats(0,"plugin.cc - malloc failure");
298 3            return EP_RB_RECOVER_MALLOC_FAILURE;
299 2          }

301 2       *temp_ptr=(char *)rn;
302 2       new_top->appData.data=(char *)temp_ptr;
303 2       new_top->appData.length=sizeof(char *);
```

```
395 2       new_top->root.objLevel=RSTRPC_tlo_type;
396 2       new_top->root.objName=esl_strdup((char *)(rn->getNameChar()));
397 2       if(NULL==new_top->root.objName)
398 3          {
399 3            rbe_log_stats(
                 0,"plugin.cc - allocation failure from esl_strdup failure");
400 3            return EP_RB_RECOVER_MALLOC_FAILURE;
401 2          }

403 2       new_top->wiBIC=NULL;
404 2       //
405 2       new_top->templateName=esl_strdup(""); // Blank as placeholder

407 2       new_top->root.objTypeString=esl_strdup("");
                                          // Blank as placeholder
408 2       new_top->hostname=esl_strdup(rcd->currentWisetListNode->getName(
                                                                    ));
409 2       new_top->fileSpec=esl_strdup(""); // Blank as placeholder

411 2       if(NULL == new_top->root.objTypeString ||
412 2          NULL == new_top->hostname ||
413 2          NULL == new_top->fileSpec)
414 3          {
415 3            rbe_log_stats(
                       0,'plugin.cc - allocation failure from esl_strdup');
416 3            return EP_RB_RECOVER_MALLOC_FAILURE;
417 2          }

419 2       new_top->wiType='\0';
420 2       new_top->ssThread=FALSE;

422 2       new_list_elem->tlo=new_top;
423 2       new_list_elem->next=NULL;

425 2       if(previous)
426 3          {
427 3            previous->next=new_list_elem;
428 2          }
429 2       else
430 3          {
431 3            *topLevelObjs=new_list_elem;
432 2          }
433 2       previous=new_list_elem;

435 1     }

437 1   return E_SUCCESS;
438   }
```



```
350   //
```

```
352  /*****************************************************
353   *  Set TopLevel Object:
354   *
355   *  This function is called to notify the plug-in that one of its top
                                                                    level
356   *  objects has been selected (for browsing and marking). The plug-in
357   *  should perform whatever validation and initialization is needed to
                                                                    prepare
358   *  for browsing and marking this top level object.
359   *  If this call returns success,
                                        then RSTPI_GetNextLevelObjects will be
360   *  called to retrieve the highest level restorable objects for this
361   *  backup object.
362   *
363   * NOTE: This function is responsible for setting the template_name
                                                                        and
364   * saveset_thread elements of the restore context.
365   *
366   * Returns:
367   *       E_SUCCESS          on success
368   *       EP_RB_RECOVER_xxx  on error
369   *
370   * Parameters:
371   *   context         (I)  - Pointer to the restore context
372   *   backupObject    (I)  - the selected top level object
373   *
374   *****************************************************************/
375  eerrno_ty RSTPI_SetTopLevelObject(restore_context *context,
376                                    struct RSTRPC_top_level_obj
                                                          *backupobject)
377  1 {
378  1   //
379  1   // Get the restore context so we can find the app data
380  1   //
381  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);
382  1
383  1   if(!rcd)
384  2     {
385  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
386  1     }
387  1
388  1   //
389  1   // Get the current node info from the top level object
390  1   //
391  1   char ** cl=(char **)(backupobject->appData.data);
392  1
393  1   RestoreNode *rnodep=(RestoreNode *)(*cl);
394  1
395  1   if(!rnodep)
396  2     {
397  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
398  1     }
399  1
400  1   //
401  1   // Top level object should always point to a work item set node
402  1   // is the case.
403  1   //
404  1   if(rnodep->nodeType() != RNC_WI_SET)
405  2     {
406  2       return EP_RB_RECOVER_TLO_NOT_WISET_NODE;
407  1     }
408  1
409  1   //
410  1   // Set the current backup node
```



```
411  1   //
412  1   rcd->currentWisetNode=(WiSetNode *)rnodep;
413  1
414  1   //
415  1   // Set the current backup node to the most recent one in time
416  1   // which is FULL
417  1   //
418  1
419  1   // Clear locks if they exist
420  1   if(NULL != rcd->currentBackupNode)
421  2     {
422  2       rcd->currentBackupNode->unlockWorkitems(context);
423  2       rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
424  1     }
425  1   rcd->currentBackupNode=NULL;
426  1
427  1   BackupNode *bnsave=NULL;
428  1   BackupNode *nextBnode;
429  1   int more=1;
430  1
431  1   for(BackupNode *bn=(BackupNode *)rnodep->getFirstChild();
432  1       more;
433  1       bn=nextBnode)
434  2     {
435  2       nextBnode=(BackupNode *)(rnodep->getNextChild());
436  1
437  2       if(NULL == nextBnode) // Force end of loop if no more items to
                                                                        scan
438  3         {
439  3           more=0;
440  2         }
441  1
442  2       //
443  2       // Populate in case this is necessary
444  2       //
445  2       // Errors will set appropriate state bits on nodes
446  2       int err=bn->populate(context, POPULATE_CHILDREN);
447  1
448  2       bnsave=bn;   // Save in case we have to return an incomplete one
449  2       if(bn->getStateBit(STATE_COMPLETE))
450  3         {
451  3           // If we already have a current backup node,
                                                        clear all marks
452  3           if(rcd->currentBackupNode)
453  4             {
454  4               rcd->currentBackupNode->unlockWorkitems(context);
455  4               rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
456  3             }
457  3           rcd->currentBackupNode=bn;
458  3           if(!rcd->currentBackupNode->lockWorkitems(context))
459  4             {
460  4               return EP_RB_RECOVER_WI_LOCKED;
461  3             }
462  3           RnProperty *tpnode=bn->getProperty(PROPERTY_TEMPLATE);
463  3           if(!tpnode)
464  4             {
465  4               return EP_RB_RECOVER_NO_TEMPLATE_PROPERTY;
466  3             }
467  3           char * templateBase=(char *)(((
                                        RnPropertyChar *)tpnode)->getValue());
468  3           backupobject->templateName=esl_strdup(templateBase);
469  3           context->rc_template_name=esl_strdup(templateBase);
470  1
471  3           if(NULL == backupobject->templateName ||
472  3              NULL == context->rc_template_name)
473  4             {
```

```
474 4                rbe_log_stats(
                       0,"plugin.cc - allocation failure from esl_strdup");
475 4                return EP_RB_RECOVER_MALLOC_FAILURE;
476 3              }

478 3            // Break out so that we get the first one on the list
479 3            // (most recent)
480 3            more=0;

482 2          }

484 1      }

486 1    //
487 1    // Check and make sure we got one
488 1    //
489 1    if(!rcd->currentBackupNode)
490 2      {
491 2        if(NULL==bnsave)
492 3          {
493 3            // We shouldn't be able to get here,
                                                but let's check for safety
494 3            rbe_log_stats(0,"Could not find any backup node");
495 3            return EP_RB_RECOVER_NONE_EXISTS;
496 2          }

498 2        if(NULL!=rcd->currentBackupNode)
499 3          {
500 3            rcd->currentBackupNode->unlockWorkitems(context);
501 2          }
502 2        rcd->currentBackupNode=bnsave;
503 2        if(!rcd->currentBackupNode->lockWorkitems(context))
504 3          {
505 3            return EP_RB_RECOVER_WI_LOCKED;
506 2          }
507 2        RnProperty *tpnode=rcd->currentBackupNode->getProperty(
                                                    PROPERTY_TEMPLATE);

508 2        if(!tpnode)
509 3          {
510 3            return EP_RB_RECOVER_NO_COMPLETE_BACKUP;
511 2          }
512 2        char * templateBase=(char *)(((
                                      RnPropertyChar *)tpnode)->getValue());
513 2        backupobject->templateName=esl_strdup(templateBase);
514 2        context->rc_template_name=esl_strdup(templateBase);

516 2        if(NULL==backupobject->templateName ||
517 2            NULL==context->rc_template_name)
518 3          {
519 3            rbe_log_stats(
                       0,"plugin.cc - allocation failure from esl_strdup");
520 3            return EP_RB_RECOVER_MALLOC_FAILURE;
521 2          }
522 1      }

524 1    return E_SUCCESS;
525    }
```



```
527    //
```

```
528 /*************************************************************
529  *  Get Next Level Objects:
530  *
531  *  This function is intended to allow retrieval of the children
532  *  of a given parent object.
                                The caller specifies the parent object and
533  *  whether or not to include bad files.  Even though the objects are
534  *  returned in a linked list, there could conceivably be thousands of
535  *  child objects, so the caller must specify the maximum number
536  *  of children to return.  The caller is returned a token (
                                                        cookie) to allow
537  *  continuing on the next call to thid function.
538  *
539  * Parameters:
540  *    context          (I)  - Pointer to the restore context
541  *    parentObject     (I)  - the parent object
542  *    objectLevel      (
                    I)  - specifies whether parentObject is a top level or
543  *                                container object
544  *    objects          (
                O)  - a pointer to receive the start of the objects list
545  *    cookie           (IO) - a place holder for the list position
546  *    maxEntries       (I)  - the maximum number of objects to return
547  *    numberEntries    (
                      O)  - the real number of objects returned in the list
548  *    allowBadFiles  (I)  - flag whether or not to include bad files
549  *
550  * Returns:
551  *       E_SUCCESS              on success
552  *       EP_RB_RECOVER_xxx      on error
553  *
554  *************************************************************************/

556 eerrno_ty RSTPI_GetNextLevelObjects(restore_context *context,
557                                     restorableObjectPtr parentObj,
558                                     enum RSTRPC_ObjectLevel
                                                        objectLevel,
559                                     struct RSTRPC_uro_list **objects,
560                                     long *cookie,
561                                     const long maxEntries,
562                                     long *numberEntries,
563                                     const boolean_ty allowBadFiles)
564 1 {

566 1   struct restoreContextData *rcd=
567 1     (struct restoreContextData *)(context->appData);

569 1   if(!rcd)
570 2     {
571 2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
572 1     }

574 1   //
575 1   // Geta the restore node of the child
576 1   //
577 1   RestoreNode *rnodep;

579 1   char ** cl;

581 1   switch(objectLevel)
582 2     {
583 2     case RSTRPC_tlo_type:
584 2       cl=(char **)(((RSTRPC_top_level_obj *)parentObj)->appData.data);
585 2       rnodep=(RestoreNode *)(*cl);
586 2       break;
587 2     case RSTRPC_container_type:
```



```
588 2     case RSTRPC_leaf_type:
589 2       cl=(char **)(((
                  RSTRPC_user_restorable_object *)parentObj)->appData.data);
590 2       rnodep=(RestoreNode *)(*cl);
591 2       break;
592 2     default:
593 2       return EP_RB_RECOVER_BAD_OBJECT_LEVEL;
594 2       break;
595 1     }

597 1   if(!rnodep)
598 2     {
599 2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
600 1     }

602 1   //
603 1   // If we are getting the children of the current workitem set node,
                                                                      we
604 1   // acrually want the children of the current backup node.
                                                  The current backup node
605 1   // is the one which pertains to the point in time we are interested
                                                                      in.
606 1   //

608 1   if(rnodep==rcd->currentWisetNode)
609 2     {
610 2       rnodep=rcd->currentBackupNode;
611 1     }

613 1   if(!rnodep)
614 2     {
615 2       return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
616 1     }


619 1   int entries_count=0;

621 1   //
622 1   // This code checks to see if we have a cookie.
                                                  If the cookie is non-null,
623 1   // it is actually a pointer to the position int he chain where we
                                                                      left
624 1   // off.
625 1   //
626 1   // Danger Will Robinson - this relies on ebing able to fit a pointer
627 1   // into a cookie (long int).
628 1   //
629 1   RestoreNode *child;
630 1   if(0 == *cookie)
631 2     {
632 2       child=rnodep->getFirstChild();
633 1     }
634 1   else
635 2     {
636 2       // Standards issue: Caution,
                                        assuming long can be cast to/from pointer
637 2       child=(RestoreNode *)*cookie;
638 1     }

640 1   RSTRPC_uro_list *previous=NULL;
641 1   RSTRPC_uro_list *listentry;
642 1   RSTRPC_user_restorable_object *urobj;


644 1   //
645 1   // Traverse the children of the current node and create a restorable
646 1   // object list.
```

```
647 1   //
648 1   for(;child;child=rnodep->getNextChild())
649 2     {
650 2       //
651 2       // Skip this entry if we have chosen not to allow bad files
652 2       //
653 2       if( (FALSE == allowBadFiles) &&
654 2           (0 != rnodep->getStateBit(STATE_BADFILE)) )
655 3           {
656 3             continue;
657 2           }

659 2       entries_count++;

661 2       listentry=(RSTRPC_uro_list *)malloc(sizeof(
                                                  struct RSTRPC_uro_list));

662 2       if(!listentry)
663 3           {
664 3             rbe_log_stats(0,"plugin.cc - malloc failure");
665 3             return EP_RB_RECOVER_MALLOC_FAILURE;
666 2           }

668 2       urobj=(RSTRPC_user_restorable_object *)malloc(sizeof(
                                    struct RSTRPC_user_restorable_object));

669 2       if(!urobj)
670 3           {
671 3             rbe_log_stats(0,"plugin.cc - malloc failure");
672 3             return EP_RB_RECOVER_MALLOC_FAILURE;
673 2           }
674 2       memset(urobj,0,sizeof(RSTRPC_user_restorable_object));

676 2       listentry->uro=urobj;

678 2       char ** temp_ptr;
679 2       temp_ptr=(char **)malloc(sizeof(char *));
680 2       if(!temp_ptr)
681 3           {
682 3             rbe_log_stats(0,"plugin.cc - malloc failure");
683 3             return EP_RB_RECOVER_MALLOC_FAILURE;
684 2           }
685 2       *temp_ptr=(char *)child;

687 2       urobj->appData.data=(char *)temp_ptr;
688 2       urobj->appData.length=sizeof(char *);

690 2       urobj->root.objLevel=child->getStateBit(
                  STATE_LEAFNODE)?RSTRPC_leaf_type:RSTRPC_container_type;

692 2       char *child_name=(char *)(child->getNameChar());
693 2       urobj->root.objName=esl_strdup(child_name);

695 2       if(NULL==urobj->root.objName)
696 3           {
697 3             rbe_log_stats(
                        0,"plugin.cc - allocation failure from esl_strdup");
698 3             return EP_RB_RECOVER_MALLOC_FAILURE;
699 2           }

701 2       // Find base name - token after final '/' if present,
                                                        else whole thing
702 2       // If the final character is a "/", look for the token preceeding
703 2       // this final character.

705 2       char *rp=NULL;
706 2       char *ch=NULL;
707 2       RnProperty *gname=child->getProperty(PROPERTY_GUINAME);
```



```
708 2       if(gname)
709 3           {
710 3             rp=(char *)((RnPropertyChar *)gname)->getValue();
711 2           }

713 2       if(NULL==rp)
714 3           {
715 3             ch=esl_strdup(child_name);
716 3             if(NULL==ch)
717 4               {
718 4                 rbe_log_stats(
                          0,"plugin.cc - allocation failure from esl_strdup");
719 4                 return EP_RB_RECOVER_MALLOC_FAILURE;
720 3               }
721 3             int len=strlen(child_name);
722 3             if(ch[len-1] == '/' && len>1)
723 4               {
724 4                 ch[len-1]='\0';
725 3               }

727 3             rp=rindex(ch,'/');

729 3             if(NULL==rp)
730 4               {
731 4                 rp=(char *)(child->getNameChar());
732 3               }
733 3             else
734 4               {
735 4                 rp++;
736 3               }
737 2           }
738 2       urobj->objBaseName=esl_strdup(rp);
739 2       if(NULL!=ch)
740 3           {
741 3             free(ch);
742 2           }

744 2       urobj->root.objTypeString=NULL; // ROB: We neeed to fill this in

746 2       char *tmp_owner=uid2str(child->getOwner());
747 2       char *tmp_group=gid2str(child->getGroup());

749 2       if(NULL==tmp_owner || NULL==tmp_group)
750 3           {
751 3             rbe_log_stats(
                          0,"plugin.cc - allocation failure from esl_strdup");
752 3             return EP_RB_RECOVER_MALLOC_FAILURE;
753 2           }

755 2       urobj->objMode=child->getMode();

757 2       //
758 2       // Archive log file node stores owner and group as character
759 2       // strings since we got it from the Network restore API
760 2       //
761 2       if(child->nodeType() == RNC_ARCHIVELOGFILE)
762 3           {
763 3             ArchivelogfileNode *anode=(ArchivelogfileNode *)(child);
764 3             urobj->objOwnerName=esl_strdup(anode->getOwnerChar());
765 3             urobj->objGroupName=esl_strdup(anode->getGroupChar());
766 2           }
767 2       else
768 3           {
769 3             urobj->objOwnerName=esl_strdup(tmp_owner);
770 3             urobj->objGroupName=esl_strdup(tmp_group);
771 2           }
```

```
773 2       if(NULL==urobj->objOwnerName || NULL==urobj->objGroupName)
774 3         {
775 3           rbe_log_stats(
                          0,"plugin.cc - allocation failure from esl_strdup");
776 3           return EP_RB_RECOVER_MALLOC_FAILURE;
777 2         }


780 2       RnPropertyDate *pr=(RnPropertyDate *)(child->getProperty(
                                                    PROPERTY_BACKUP_DATE));
781 2       if(pr)
782 3         {
783 3           urobj->objModTime=pr->getValue()->unixTime();
784 2         }
785 2       else
786 3         {
787 3           urobj->objModTime=0;
788 3           BackupNode *bnode=(BackupNode *)(child->getBackupNodePtr());
789 3           if(bnode)
790 4             {
791 4               pr=(RnPropertyDate *)(bnode->getProperty(
                                                    PROPERTY_BACKUP_DATE));
792 4               if(pr)
793 5                 {
794 5                   urobj->objModTime=pr->getValue()->unixTime();
795 4                 }
796 3             }
797 2         }

799 2       urobj->objSize.high=child->getSize().high;
800 2       urobj->objSize.low=child->getSize().low;

802 2       if(child->getStateBit(STATE_BADFILE))
803 3         {
804 3           urobj->objBackupStatus=RSTRPC_Backup_Bad;
805 2         }
806 2       else if( child->getStateBit(STATE_INVALID_SSID |
807 2                                   STATE_NO_SSID      |
808 2                                   STATE_EXPIRED_SSID  ) )
809 3         {
810 3           urobj->objBackupStatus=RSTRPC_Backup_Expired;
811 2         }
812 2       else if( child->getStateBit(STATE_CH_INVALID_SSID |
813 2                                   STATE_CH_NO_SSID      |
814 2                                   STATE_CH_EXPIRED_SSID |
815 2                                   STATE_CH_BADFILE       )  )
816 3         {
817 3           urobj->objBackupStatus=RSTRPC_Backup_Child_Without_Data;
818 2         }
819 2       else
820 3         {
821 3           urobj->objBackupStatus=RSTRPC_Backup_Good;
822 2         }

824 2       listentry->next=NULL;

826 2       if(previous)
827 3         {
828 3           previous->next=listentry;
829 2         }
830 2       else
831 3         {
832 3           *objects=listentry;
833 2         }
834 2       previous=listentry;
```



```
836 2       if(entries_count==maxEntries)
837 3         {
838 3           *cookie=(long)child;
839 3           *numberEntries=entries_count;
840 3           return E_SUCCESS;
841 2         }
842 1     }

844 1     *cookie=DONE_COOKIE;
845 1     *numberEntries=entries_count;
846 1     return E_SUCCESS;
847   }
```

```
849 //
```



```
851 /*****************************************************************
852  * Mark Object
853  *
854  *   The MarkObject operation takes a restorableObject and marks it, and
855  *   possibly its descendant files for restoral based on the input criteria.
856  *   Since the RSTSL_MarkObject call is an asynchronously executed operation
857  *   in the Restore Engine that performs the marking, this function must
858  *   periodically check for user-signalled cancelation, and update progress
859  *   data using the progress callback function argument.
860  *
861  * NOTE: This functions is responsible for keeping the volumes needed list
862  *       (ebvllist) element of the restore context up to date.
863  *
864  * Parameters:
865  *
866  *  context         (I) - Pointer to the restore context
867  *  thisObject    (I) - The restoral object; can be a leaf object (e.g. a
868  *                      file), or a container object (e.g., a directory).
869  *  allowBadfiles (I) - allows marking of files of state BADDATA.
870  *  descend         (I) - Should mark operation descend to operate on the content
871  *                        of container objects.
872  *  BadFilesCount (O) - returns the file count with BADDATA.
873  *  PermDenyFilesCount (O) -- returns the file count with permission denied.
874  *  fileMarked      (O) - return the total files marked after this mark occurred.
875  *  lenMarkedFiles(O) - return the length of files marked after this mark
876  *  dirMarked       (O) - return the total directories marked after this mark
877  *                        occurred.
878  *  otherMarked     (O) - return the total "other" files marked after this mark.
879  *  progressCB      (I) - pointer to callback function to report progress and
880  *                        test for cancellation
881  *
882  *  Note:
883  *         The Progress Callback is currently not used.  If the need
884  *         for this developes, the routines which use this argument
885  *         (mark, unmark, submit) must be enhanced.
886  *
887  *
888  *****************************************************************/


891 eerrno_ty RSTPI_MarkObject(restore_context *context,
892                            struct RSTRPC_user_restorable_object *thisObject,
893                            boolean_ty allowBadFiles,
895                            boolean_ty descend,
896                            unsigned long *badFilesCount,
897                            unsigned long *permDenyFilesCount,
898                            unsigned long *filesMarked,
899                            u_hyper *lenMarkedFiles,
```

```
900                                     unsigned long *dirMarked,
901                                     unsigned long *otherMarked,
902                                     RSTPI_MarkProgressProc progressCB)
903 1 {
904 1   char ** cl=(char **)(thisObject->appData.data);
905 1   RestoreNode *rnodep=(RestoreNode *)(*cl);

907 1   if(NULL==rnodep)
908 2     {
909 2       rbe_log_stats(
                        0,"RSTPI_MarkObject - Mark object has no app data");
910 2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
911 1     }

913 1   /*
914 1    * MTFB (multi trail file backup) nodes are to be treated as atomic.
915 1    * If the item we are trying to mark is a child of a MTFB node, then
916 1    * we perform the mark operation on the entire MTFB node.
917 1    */
918 1   RestoreNode *parent=rnodep->getParent();
919 1   if(NULL !=parent  && parent->nodeType() == RNC_MTFB)
920 2     {
921 2       rnodep=parent;
922 1     }

924 1   *badFilesCount=0;
925 1   *permDenyFilesCount=0;
926 1   *filesMarked=0;
927 1   *lenMarkedFiles=ul_to_uh(0);
928 1   *dirMarked=0;
929 1   *otherMarked=0;

931 1   rnodep->markNode(descend,allowBadFiles);

933 1   // We start at the backup node pointer since we want the total
934 1   // for the entire backup, not just this node down (except the
935 1   // badFilesCount, in which case the affected nodes will only
936 1   // be from this level down)

938 1   *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                    RNC_ANY_DATAFILE);
939 1   *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();


942 1   // The bad file count is a cound of the bad files encountered
943 1   // (not necessarily marked) from the current node down.

945 1   *badFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

947 1   return E_SUCCESS;
948   }
```



```
950 //
```

```
951  /*****************************************************************
952   * UnmarkObject
953   *
954   *   The UnmarkObject operation takes a restorableObject and unmarks
                                                              it, and
955   *   possibly its descendant files for restoral based on the input
                                                              criteria.
956   *   Since the RSTSL_UnmarkObject call is an asynchronously executed
                                                              operation
957   *   in the Restore Engine that performs the unmarking,
                                                  this plug-in function
958   *   must periodically check for user-signalled cancelation,
                                                          and update
959   *   progress data using the progress callback function argument.
960   *
961   * NOTE: This functions is responsible for keeping the volumes needed
                                                              list
962   *       (ebvllist) element of the restore context up to date.
963   *
964   * UnmarkObject Parameters:
965   *
966   *  context         (I) - Pointer to the restore context
967   *  thisObject    (I) - The restoral object;
                                        can be a leaf object (e.g. a
968   *                        file), or a container object (
                                        e.g., a directory).
969   *  BadFilesOnly  (
                        I) - allows unmarking ONLY of files of state BADDATA.
970   *  descend       (
                        I) - Should unmark operation descend to operate on the
971   *                        content of container objects.
972   *  BadFilesCount (O) - returns the file count with BADDATA.
973   *  fileMarked    (
                        O) - return the total files marked after this unmark
974   *  lenMarkedFiles(
                    O) - return the length of files marked after this unmark
975   *  dirMarked     (
                    O) - return the total directories marked after this unmark
976   *  otherMarked   (
                    O) - return the total "other" files marked after this unmark
977   *  progressCB    (
                    I) - pointer to callback function to report progress and
978   *                        test for cancellation
979   *  Note:
980   *         The Progress Callback is currently not used.  If the need
981   *         for this developes, the routines which use this argument
982   *         (mark, unmark, submit) must be enhanced.
983   *
984   *****************************************************************/


987  eerrno_ty RSTPI_UnmarkObject(restore_context *context,
988                               struct RSTRPC_user_restorable_object
                                                          *thisObject,
989                               const boolean_ty BadFiledOnly,
990                               const boolean_ty descend,
991                               unsigned long *BadFilesCount,
992                               unsigned long *filesMarked,
993                               u_hyper *lenMarkedFiles,
994                               unsigned long *dirMarked,
995                               unsigned long *otherMarked,
996                               RSTPI_MarkProgressProc progressCB)
997  1 {
998  1   *BadFilesCount=0;
999  1   *filesMarked=0;
1000 1   *lenMarkedFiles=ul_to_uh(0);
```



```
1001 1   *dirMarked=0;
1002 1   *otherMarked=0;

1004 1   char ** c1=(char **)(thisObject->appData.data);
1005 1   RestoreNode *rnodep=(RestoreNode *)(*c1);

1007 1   if(NULL==rnodep)
1008 2     {
1009 2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1010 1     }


1013 1   RestoreNode *parent=rnodep->getParent();
1014 1   if(NULL !=parent  && parent->nodeType() == RNC_MTFB)
1015 2     {
1016 2       rnodep=parent;
1017 1     }
1018 1   /*
1019 1    * MTFB (multi trail file backup) nodes are to be treated as atomic.
1020 1    * If the item we are trying to unmark is a child of a MTFB node,
                                                              then
1021 1    * we perform the unmark operation on the entire MTFB node.
1022 1    */


1025 1   rnodep->unmarkNode(descend,FALSE);

1027 1   // We start at the backup node pointer since we want the total
1028 1   // for the entire backup, not just this node down (except the
1029 1   // badFilesCount, in which case the affected nodes will only
1030 1   // be from this level down)

1032 1   *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                          RNC_ANY_DATAFILE);
1033 1   *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();

1035 1   // The bad file count is a cound of the bad files encountered
1036 1   // (not necessarily unmarked) from the current node down.

1038 1   *BadFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

1040 1   return E_SUCCESS;

1042   }
```

```
1046  //
```



```
1047  /*****************************************************************************
1048   * Is Object Markable
1049   *
1050   *  This function determines if a restorable object has been
1051   *  marked for restoration.                    It is intended to allow the user to determine the
1052   *  current restore markings for the restorable objects at the same                                object tree
1053   *  hierarchy level,
                  i.e. objects that have the same parent restorableObject.
1054   *
1055   * Parameters:
1056   *
1057   * context       (I) - Pointer to the restore context
1058   * thisObject    (
              I) - The restoral object to be checked: can be a leaf object
1059   *                        (e.g. a file), or a container object (                                    e.g., a directory).
1060   * markable      (O) - boolean to receive the marked(1) / unmarked(                                0) result
1061   *****************************************************************************/

1063  eerrno_ty RSTPI_IsObjectMarkable(restore_context *context,
1064                                   struct RSTRPC_user_restorable_object                                    *thisObject,
1065                                   boolean_ty *markable)
1066  1 {

1068  1   RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);

1070  1   if(NULL==rnodep)
1071  2     {
1072  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1073  1     }

1075  1   *markable=rnodep->isNodeMarkable();

1077  1   return E_SUCCESS;

1079    }
```

```
1082  //
```



```
1083  /*****************************************************************
1084   * Is Object Marked
1085   *
1086   *  This function determines if a restorable object has been
1087   *  marked for restoration.
                                    It is intended to allow the user to determine the
1088   *  current restore markings for the restorable objects at the same
                                                                    object tree
1089   *  hierarchy level,
                    i.e. objects that have the same parent restorableObject.
1090   *
1091   * Parameters:
1092   *
1093   * context       (I) - Pointer to the restore context
1094   * thisObject   (
              I) - The restoral object to be checked: can be a leaf object
1095   *                      (e.g. a file), or a container object (
                                                          e.g., a directory).
1096   * marked        (O) - boolean to receive the marked(1) / unmarked(
                                                                    0) result
1097   *****************************************************************/

1099  eerrno_ty RSTPI_IsObjectMarked(restore_context *context,
1100                                 struct RSTRPC_user_restorable_object
                                                                  *thisObject,
1101                                 boolean_ty *marked)
1102  1 {

1104  1   RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);

1106  1   if(NULL==rnodep)
1107  2     {
1108  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1109  1     }

1111  1   *marked=rnodep->isNodeMarked();

1113  1   return E_SUCCESS;

1115      }
```

```
1117    //
```



```
1118    /*****************************************************************
                                                                        **
1119    *
1120    * Get All Backup Times
1121    *
1122    * Function Description:
1123    *      Retrieve the dates of the backups within the time range
1124    *      specified by the caller.
1125    *
1126    * Parameters:
1127    *      context                 - (I) Pointer to the restore context
1128    *      startTime       - (I) include no earlier than this date
1129    *      endTime         - (I) include no later than this date
1130    *      flags           - (I) flags - complete/partial
1131    *      timesList       - (O) ptr to linked list of times
1132    *      numEntries      - (O) count of times returned
1133    *
1134    * Return Codes:
1135    *      E_SUCCESS       - operation completed successfully
1136    *
1137    *****************************************************************/

1139    eerrno_ty RSTPI_GetAllBackupTimes(restore_context *context,
1140                                      const time_t startTime,
1141                                      const time_t endTime,
1142                                      RSTRPC_backup_flags_ty flags,
1143                                      RSTRPC_time_list **timesList,
1144                                      short *numEntries)
1145  1 {

1147  1   *timesList=NULL;

1149  1   //
1150  1   // Now look through current backups and put on list if in
1151  1   // Range
1152  1   //
1153  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);


1155  1   if(!rcd)
1156  2     {
1157  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1158  1     }

1160  1   WiSetNode *wilist=rcd->currentWisetNode;

1162  1   if(!wilist)
1163  2     {
1164  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1165  1     }


1168  1    int first=1;

1170  1    RSTRPC_time_list *newlist=NULL;

1172  1    *numEntries=0;     // Count of number of entries found

1174  1    for(RestoreNode *bn=wilist->getFirstChild();bn;
                                                bn=wilist->getNextChild())
1175  2      {
1176  2        RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
                                                    PROPERTY_BACKUP_DATE));

1177  2        if(!pr)
1178  3          {
1179  3            continue;
```

```
1180 2          }
1181 2        // Get Unix flavored time

1183 2        DateTime *dt=(DateTime *)(pr->getValue());

1185 2        time_t backupTime=dt->unixTime();

1187 2        // time_t backupTime=(DateTime)(pr->getValue())->unixTime();
1188 2        //
1189 2        // Get all dates if start and end time are zero
1190 2        // otherwise make sure the time is in the selected range
1191 2        //
1192 2        // Also, get complete backups ONLY if that is what is required
1193 2        //
1194 2        //
1195 2        if (
1196 2              ( ( (startTime==0) && (endTime==0) ) ||
1197 2                (startTime<=backupTime) && (endTime>=backupTime)
1198 2              )
1199 2               &&
1200 2              ( (flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) ||
1201 2                bn->getStateBit(STATE_COMPLETE)
1202 2              )
1203 2            )
1204 3               {
1205 3                 //
1206 3                 // Ok, we have found a backup time within the
1207 3                 // range.  Add it here.
1208 3                 //
1209 3                 RSTRPC_time_list *newtime=(RSTRPC_time_list *)malloc(
                                          sizeof(struct RSTRPC_time_list));
1210 3                 if(!newtime)
1211 4                   {
1212 4                     rbe_log_stats(0,"plugin.cc - malloc failure");
1213 4                     return EP_RB_RECOVER_MALLOC_FAILURE;
1214 3                   }

1216 3                 //
1217 3                 // Increment count of valid time entries
1218 3                 //
1219 3                 (*numEntries)++;

1221 3                 newtime->next=NULL;
1222 3                 newtime->time=backupTime;
1223 3                 if(newlist)
1224 4                   {
1225 4                     newlist->next=newtime;
1226 3                   }
1227 3                 newlist=newtime;

1229 3                 if(first)
1230 4                   {
1231 4                     *timesList=newlist;
1232 4                     first=0;
1233 3                   }

1235 2               }
1236 1      }

1238 1    return E_SUCCESS;
1239     }
```



```
1241    //
```

```
1242  /*****************************************************************************
                                                                              **
1243   *
1244   * RSTSL_GetCurrentBackupTime
1245   *
1246   * Function Description:
1247   *      Retrieve the time of the backup that the current restore
                                                                    context
1248   *      is set to and return it in the preallocated buffer.
1249   *
1250   * Parameters:
1251   *       context          - (I) Pointer to the restore context
1252   *      bkupTime - (O) the time of the backup
1253   *
1254   * Return Codes:
1255   *      E_SUCCESS        - operation completed successfully
1256   *      EP_RB_RECOVER_INVALOP   - call issued out of sequence
1257   *      EP_RB_RECOVER_BAD_ARGS  - invalid input argument
1258   *      EP_RB_RECOVER_NO_CURR_BACKUP     - no valid backup currently
1259   *
1260   *****************************************************************************/

1262  eerrno_ty RSTPI_GetCurrentBackupTime(restore_context *context,
1263                                       time_t *backupTime)
1264  1 {
1265  1   //
1266  1   // Now look through current backups and put on list if in
1267  1   // Range
1268  1   //
1269  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);

1271  1   if(!rcd)
1272  2     {
1273  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1274  1     }

1276  1   BackupNode *currentBackup=rcd->currentBackupNode;
1277  1   if(!currentBackup)
1278  2     {
1279  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1280  1     }

1282  1   RnPropertyDate *pr=(RnPropertyDate *)(currentBackup->getProperty(
                                                    PROPERTY_BACKUP_DATE));
1283  1   if(!pr)
1284  2     {
1285  2       *backupTime=0;
1286  2       return EP_RB_RECOVER_NO_BACKUPTIME;
1287  1     }

1289  1   DateTime *dt=(DateTime *)(pr->getValue());
1290  1   *backupTime=dt->unixTime();

1292  1   return E_SUCCESS;
1293  }
```



```
1297  //
```

```
1298  /*****************************************************************************
                                                                              **
1299  *
1300  * Set Backup For Time
1301  *
1302  * Function Description:
1303  *       Switch to the backup plane of the specified time,
                                                          or the most recent
1304  *         that is before the specified time,
                                          if an exact mattch is not possible.
1305  *
1306  * Parameters:
1307  *  context (I) - Pointer to the restore context
1308  *  forTime (I) - The time for which the backup is requested
1309  *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
1310  *
1311  * Return Codes:
1312  *       E_SUCCESS                  - operation completed successfully
1313  *       EP_RB_RECOVER_xxx          - backup plane cannot be found
1314  *
1315  *****************************************************************************/

1317  eerrno_ty RSTPI_SetBackupForTime(restore_context *context,
1318                                   const time_t forTime,
1319                                   RSTRPC_backup_flags_ty flags)

1321 1 {

1323 1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);

1325 1   if(!rcd)
1326 2     {
1327 2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1328 1     }

1330 1   WiSetNode *wilist=rcd->currentWisetNode;

1332 1   if(!wilist)
1333 2     {
1334 2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1335 1     }



1338 1   //
1339 1   // Loop through each one on the current list and return the one
                                                              which we
1340 1   // are looking for.
1341 1   //
1342 1   for(RestoreNode *bn=wilist->getFirstChild();bn;
                                                   bn=wilist->getNextChild())

1343 2     {
1344 2       RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
                                                    PROPERTY_BACKUP_DATE));
1345 2       if(!pr)
1346 3         {
1347 3           continue;
1348 2         }
1349 2       // Get Unix flavored time

1351 2       DateTime *dt=(DateTime *)(pr->getValue());

1353 2       time_t backupTime=dt->unixTime();

1355 2       if ( (forTime >= backupTime) &&
1356 2            ( (
```



```
                flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || bn->getStateBit(
                                                          STATE_COMPLETE) )
1357 2         )
1358 3         {
1359 3           BackupNode *bnode=(BackupNode *)bn;

1361 3           // If we already have a current backup node which differs
                                                                  from the
1362 3           // current, clear marks.
1363 3           //
1364 3           if(
      NULL != rcd->currentBackupNode && rcd->currentBackupNode != bnode)
1365 4             {


1368 4               rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1369 3             }
1370 3           if(NULL!=rcd->currentBackupNode)
1371 4             {
1372 4               rcd->currentBackupNode->unlockWorkitems(context);
1373 3             }
1374 3           rcd->currentBackupNode=bnode;
1375 3           if(!rcd->currentBackupNode->lockWorkitems(context))
1376 4             {
1377 4               return EP_RB_RECOVER_WI_LOCKED;
1378 3             }
1379 3           bnode->populate(context,POPULATE_CHILDREN);
1380 3           // We found the one to backup
1381 3           return E_SUCCESS;
1382 2         }
1383 1     }

1385 1   // Commented out to leave current backup unchanged if no call found

1387 1   // rcd->currentBackupNode=NULL;

1389 1   return E_RB_RECOVER_NO_BACKUP_FOUND;

1391  }
```

```
1394  //
```



```
1395  /*****************************************************************
                                                                      **
1396   *
1397   * Set Previous Backup
1398   *
1399   * Function Description:
1400   *      Set the restore context to that of the previous backup with
                                                                respect
1401   *      to the current one.
1402   *
1403   * Parameters:
1404   *  context (I) - Pointer to the restore context
1405   *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
1406   *
1407   * Return Codes:
1408   *      E_SUCCESS       - operation completed successfully
1409   *      EP_RB_NO_PREV_CATALOG - when at the first catalog
1410   *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                     file
1411   *                                        of the new catalog
1412   *
1413   *****************************************************************/

1415  eerrno_ty RSTPI_SetPrevBackup(restore_context *context,
1416                                RSTRPC_backup_flags_ty flags)
1417  1 {

1419  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                    context->appData);



1422  1   WiSetNode *wilist=rcd->currentWisetNode;

1424  1   if(!wilist)
1425  2     {
1426  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1427  1     }


1430  1   BackupNode *bnode=rcd->currentBackupNode;

1432  1   if(!bnode)
1433  2     {
1434  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1435  1     }

1437  1   //
1438  1   // Set the iterator to the curent item
1439  1   //

1441  1   if(!wilist->setIterToSpecificChild(rcd->currentBackupNode))
1442  2     {
1443  2       return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
1444  1     }

1446  1   bool first=1;
1447  1   for(RestoreNode *rn=wilist->getThisChild();rn;
                                                rn=wilist->getPrevChild())
1448  2     {
1449  2       // Skip this node so we don't get self
1450  2       if(first)
1451  3           {
1452  3             first=0;
1453  2           }
1454  2       else
1455  3           {
```

```
1457 3            if( (
                 flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                              STATE_COMPLETE))
1458 4              {
1460 4                // If we already have a current backup node,
                                                          clear all marks
1461 4                if(rcd->currentBackupNode)
1462 5                  {
1463 5                    rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1464 4                  }
1465 4                if(NULL!=rcd->currentBackupNode)
1466 5                  {
1467 5                    rcd->currentBackupNode->unlockWorkitems(context);
1468 4                  }
1469 4                BackupNode *bnode=(BackupNode *)rn;
1470 4                rcd->currentBackupNode=bnode;
1471 4                if(!rcd->currentBackupNode->lockWorkitems(context))
1472 5                {
1473 5                  return EP_RB_RECOVER_WI_LOCKED;
1474 4                }
1475 4                bnode->populate(context,POPULATE_CHILDREN);
1476 4                // We found the one to backup
1477 4                return E_SUCCESS;
1478 3              }
1479 2          }
1480 1      }

1482 1    return EP_RB_RECOVER_NONE_EXISTS;
1483      }
```



```
1487  //
```

```
1488  /*****************************************************************************
                                                                              **
1489   *
1490   * Set Next Backup
1491   *
1492   * Function Description:
1493   *  This routine must set the restore environment to the the next
                                                                    backup
1494   *  of the current top level object.
1495   *
1496   *  Parameters:
1497   *       context (I) - Pointer to the restore context
1498   *       flags   (
                  I) - Backup constraint flags: e.g., full-only/partial-ok
1499   *
1500   * Return Codes:
1501   *       E_SUCCESS        - operation completed successfully
1502   *       EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent
                                                                    catalog
1503   *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                    file
1504   *                                        of the new catalog
1505   *       EP_RB_RECOVER_NO_CATALOG - when mcat_set_mcplane failed
1506   *
1507   *****************************************************************************/

1509  eerrno_ty RSTPI_SetNextBackup(restore_context *context,
1510                                RSTRPC_backup_flags_ty flags)
1511  1 {

1513  1   struct restoreContextData *rcd=(struct restoreContextData *){
                                                        context->appData);



1516  1   WiSetNode *wilist=rcd->currentWisetNode;

1518  1   if(!wilist)
1519  2     {
1520  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1521  1     }


1524  1   BackupNode *bnode=rcd->currentBackupNode;

1526  1   if(!bnode)
1527  2     {
1528  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1529  1     }


1532  1   RestoreNode *lastprev=NULL;


1535  1   RestoreNode *rn=wilist->getFirstChild();
1536  1   while(rn && rn != bnode)
1537  2     {
1538  2       if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                          STATE_COMPLETE))

1539  3           {
1540  3             lastprev=rn;
1541  2           }
1542  2       rn=wilist->getNextChild();
1543  1     }


1546  1   if(lastprev)
```



```
1547  2   {
1548  2     // If we already have a current backup node, clear all marks
1549  2     if(rcd->currentBackupNode)
1550  3         {


1552  3           rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1553  2         }
1554  2     if(NULL!=rcd->currentBackupNode)
1555  3         {
1556  3           rcd->currentBackupNode->unlockWorkitems(context);
1557  2         }
1558  2     BackupNode *bnode=(BackupNode *)lastprev;
1559  2     rcd->currentBackupNode=bnode;
1560  2     if(!rcd->currentBackupNode->lockWorkitems(context))
1561  3     {
1562  3       return EP_RB_RECOVER_WI_LOCKED;
1563  2     }
1564  2     bnode->populate(context,POPULATE_CHILDREN);
1565  2     // We found the one to backup

1567  2     return E_SUCCESS;
1568  1   }

1570  1 return EP_RB_RECOVER_NONE_EXISTS;
1571    }
```

```
1578  //
```



```
1579  /*****************************************************************
                                                                    **
1580  *
1581  * Set Most Recent Backup
1582  *
1583  * Function Description:
1584  *      Set the restore context to that of the most recent backup
                                                              catalog
1585  *      plane.
1586  *
1587  * Parameters:
1588  *       context (I) - Pointer to the restore context
1589  *       flags   (
                  I) - Backup constraint flags: e.g., full-only/partial-ok
1590  *a
1591  * Return Codes:
1592  *      E_SUCCESS        - operation completed successfully
1593  *
1594  *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                 file of
1595  *                                          the new catalog.
1596  *
1597  *****************************************************************/

1599  eerrno_ty RSTPI_SetMostRecentBackup(restore_context *context,
                                    RSTRPC_backup_flags_ty flags)
1601 1  {

1603 1    struct restoreContextData *rcd=(struct restoreContextData *)(
                                                  context->appData);



1606 1    WiSetNode *wilist=rcd->currentWisetNode;

1608 1    if(!wilist)
1609 2      {
1610 2        return EP_RB_RECOVER_RC_APP_DATA_NULL;
1611 1      }

1613 1    RestoreNode *bnode=NULL;

1615 1    //
1616 1    // Set the iterator to the curent item
1617 1    //

1619 1    for(RestoreNode *rn=wilist->getFirstChild();rn;
                                                 rn=wilist->getNextChild())
1620 2      {
1621 2        if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                   STATE_COMPLETE))

1622 3            {
1623 3              // If we already have a current backup node,
                                                          clear all marks
1624 3              if(rcd->currentBackupNode)
1625 4                {
1626 4                  rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1627 3                }
1628 3              if(NULL != rcd->currentBackupNode)
1629 4                {
1630 4                  if(rn==rcd->currentBackupNode)
1631 5                    {
1632 5                      /* We are already at the right place so return
                                                              success */

1633 5                      return E_SUCCESS;
1634 4                    }
1635 4                  rcd->currentBackupNode->unlockWorkitems(context);
```

```
1636 3              }
1637 3              BackupNode *bnode=(BackupNode *)rn;
1638 3              rcd->currentBackupNode=bnode;
1639 3              if(!rcd->currentBackupNode->lockWorkitems(context))
1640 4                {
1641 4                  return EP_RB_RECOVER_WI_LOCKED;
1642 3                }
1643 3              bnode->populate(context,POPULATE_CHILDREN);
1644 3              // We found the one to backup

1646 3              return E_SUCCESS;
1647 2            }
1648 1      }


1651 1    return EP_RB_RECOVER_SET_BUTIME_ERROR;
1652      }
```



```
1657    //
```

```
1659  /*****************************************************************************
                                                                              **
1660   *
1661   * Set First Backup
1662   *
1663   * Function Description:
1664   *       Set the restore context to that of the first backup plane.
1665   *
1666   * Parameters:
1667   *       context (I) - Pointer to the restore context
1668   *       flags   (
                    I) - Backup constraint flags: e.g., full-only/partial-ok
1669   *
1670   * Return Codes:
1671   *       E_SUCCESS         - operation completed successfully
1672   *
1673   *       EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                    file of
1674   *                                        the new catalog
1675   *
1676   *****************************************************************************/

1678  eerrno_ty RSTPI_SetFirstBackup(restore_context *context,
1679                                 RSTRPC_backup_flags_ty flags)
1680  1 {

1682  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                    context->appData);



1685  1   WiSetNode *wilist=rcd->currentWisetNode;

1687  1   if(!wilist)
1688  2     {
1689  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1690  1     }

1692  1   //
1693  1   // Set the iterator to the curent item
1694  1   //

1696  1   for(RestoreNode *rn=wilist->getFirstChild();rn;
                                          rn=wilist->getNextChild())
1697  2     {
1698  2       if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                              STATE_COMPLETE))
1699  4         {
1700  3           // If we already have a current backup node,
                                                    clear all marks
1701  3           if(rcd->currentBackupNode)
1702  4             {
1703  4               rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1704  3             }
1705  3           if(NULL!=rcd->currentBackupNode)
1706  4             {
1707  4               rcd->currentBackupNode->unlockWorkitems(context);
1708  3             }
1709  3           BackupNode *bnode=(BackupNode *)rn;
1710  3           rcd->currentBackupNode=bnode;
1711  3           if(!rcd->currentBackupNode->lockWorkitems(context))
1712  4             {
1713  4               return EP_RB_RECOVER_WI_LOCKED;
1714  3             }
1715  3           bnode->populate(context,POPULATE_CHILDREN);
```



```
1717  3           return E_SUCCESS;
1718  2         }
1719  1     }

1721  1   return EP_RB_RECOVER_SET_BUTIME_ERROR;
1722      }
```

```
1728  //
```



```
1730  /*****************************************************************
                                                                      *
1731   * Is There Prev Backup
1732   *
1733   * Function Description:
1734   *       Determine if a backup exists prior to the backup that is
1735   *       currently selected.
1736   *
1737   * Parameters:
1738   *       context (I) - Pointer to the restore context
1739   *       flags   (
                    I) - Backup constraint flags: e.g., full-only/partial-ok
1740   *       isThere  (O) - TRUE/FALSE that requested backup does exist
1741   *
1742   * Return Codes:
1743   *       E_SUCCESS                - operation completed successfully
1744   *       EP_RB_RECOVER_xxx        - when errors occur accessing
                                                                catalogs
1745   *
1746   *****************************************************************
                                                                      */

1748  eerrno_ty RSTPI_IsTherePrevBackup(restore_context *context,
1749                                    RSTRPC_backup_flags_ty flags,
1750                                    boolean_ty *isThere)
1751  1 {

1753  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);


1755  1   *isThere=FALSE;

1757  1   WiSetNode *wilist=rcd->currentWisetNode;

1759  1   if(!wilist)
1760  2     {
1761  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1762  1     }


1765  1   BackupNode *bnode=rcd->currentBackupNode;

1767  1   //Debuggine only
1768  1   RestoreNode *rob=bnode;

1770  1   if(!bnode)
1771  2     {
1772  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1773  1     }

1775  1   //
1776  1   // Set the iterator to the curent item
1777  1   //

1779  1   if(!wilist->setIterToSpecificChild(bnode))
1780  2     {
1781  2       return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
1782  1     }

1784  1   bool first=TRUE;

1786  1   for(RestoreNode *rn=wilist->getThisChild();rn;
                                                  rn=wilist->getPrevChild())
1787  2     {
1788  2       if(first)
```

```
1789  3          {
1790  3            first=FALSE;
1791  2          }
1792  2        else
1793  3          {
1794  3            if(
                   flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                              STATE_COMPLETE))
1795  4              {
1796  4                *isThere=TRUE;
1797  4                return E_SUCCESS;
1798  3              }
1799  2          }
1800  1      }

1802  1    return E_SUCCESS;
1803       }
```



```
1807  //
```

```
1808  /*****************************************************************************
                                                                                *
1809   * Is There Next Backup
1810   *
1811   * Function Description:
1812   *       Determine if a backup exists after the backup that is
1813   *       currently selected.
1814   *
1815   * Parameters:
1816   *       context (I) - Pointer to the restore context
1817   *       flags   (
                    I) - Backup constraint flags: e.g., full-only/partial-ok
1818   *       isThere  (O) - TRUE/FALSE that requested backup does exist
1819   *
1820   * Return Codes:
1821   *       E_SUCCESS                  - operation completed successfully
1822   *       EP_RB_RECOVER_xxx          - when errors occur accessing
                                                                    catalogs
1823   *
1824   *****************************************************************************
                                                                                */

1826  eerrno_ty RSTPI_IsThereNextBackup(restore_context *context,
1827                                    RSTRPC_backup_flags_ty flags,
1828                                    boolean_ty *isThere)
1829  1 {
1830  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);



1833  1   *isThere=FALSE;

1835  1   WiSetNode *wilist=rcd->currentWisetNode;

1837  1   if(!wilist)
1838  2     {
1839  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1840  1     }


1843  1   BackupNode *bnode=rcd->currentBackupNode;

1845  1   if(!bnode)
1846  2     {
1847  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
1848  1     }



1852  1   RestoreNode *rn=wilist->getFirstChild();
1853  1   while(rn && rn != bnode)
1854  2     {
1855  2       if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                          STATE_COMPLETE))
1856  3         {
1857  3           *isThere=TRUE;
1858  3           return E_SUCCESS;
1859  2         }
1860  2       rn=wilist->getNextChild();
1861  1     }


1864  1   return E_SUCCESS;


1866  }
```



```
1869  /*****************************************************************
1870   * Is Object Markable
1871   *
1872   * Function Description:
1873   *       Returns TRUE if the specified object is markable by the user,
1874   *       returns FALSE if it is not. This function applies only to
1875   *       container (directory) and leaf (file) objects.
1876   *
1877   * Parameters:
1878   *        context              - (I) Pointer to the restore context
1879   *       thisObject - (I) ptr to the restorableObject in question
1880   *
1881   * Return:
1882   *       TRUE - the specified object is markable by the user;
1883   *       FALSE - the specified object is not markable by the user;
1884   *
1885   *****************************************************************/
1886  boolean_ty
1887  RSTPI_IsObjectMarkable(
                                restore_context                 *context,
1888                               struct RSTRPC_user_restorable_object
                                                                *thisObject )
1889  1 {
1890  1       char **cl=(char **)(thisObject->appData.data);
1891  1       RestoreNode *rnodep=(RestoreNode *)(*cl);
1892  1       return rnodep->isNodeMarkable();

1894  1       return E_SUCCESS;
1895  }
```

```
1898
1899  /*****************************************************************
1900   * Get Top Level Templates:
1901   *
1902   *  This function is required to retrieve the templates with which a
                                                                    backup
1903   *  object could have been backed up.
1904   *
1905   * Parameters:
1906   *    context         (I)  - Pointer to the restore context
1907   *    topLevelObj     (I)  - the top level object
1908   *    templates       (
                O)  - pointer to receive the start of the list of templates
1909   *    numberEntries (
                O)  - the real number of templates returned in the list
1910   *
1911   *****************************************************************/
1912  eerrno_ty RSTPI_GetTopLevelTemplates(
                                        restore_context             *context,
1913                                   struct RSTRPC_top_level_obj *topLevelObj,
1914                                  struct RSTRPC_name_list      **templates,
1915                                  short
                                                          *numberEntries )
1916  1 {
1917  1   char ** cl=(char **)(topLevelObj->appData.data);
1918  1   RestoreNode *rnodep=(RestoreNode *)(*cl);
1919  1
1920  1   if(NULL==rnodep)
1921  2     {
1922  2       rbe_log_stats(
                0,"RSTPI_GetTopLevelTemplates - Mark object has no app data");
1923  2       return EP_RB_RECOVER_RN_APP_DATA_NULL;
1924  1     }
1925  1
1926  1   //
1927  1   // Since we are setting a top level object,
                                                    the marks on any existing
1928  1   // object must be cleared.
1929  1   //
1930  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);
1931  1   if(NULL!=rcd)
1932  2     {
1933  2       if(NULL!=rcd->currentBackupNode)
1934  3           {
1935  3             rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
1936  2           }
1937  1     }
1938
1939
1940  1   *numberEntries=NULL;
1941  1   *templates=NULL;
1942
1943
1944  1   for(RestoreNode *rn=rnodep->getFirstChild();
1945  1       rn;
1946  1       rn=rnodep->getNextChild())
1947  2     {
1948
1949  2       if(rn->nodeType() != RNC_BACKUP)
1950  3           {
1951  3             continue;
1952  2           }
```



```
1954  2       RnProperty *tp=rn->getProperty(PROPERTY_TEMPLATE);
1955  2       if(NULL==tp)
1956  3       {
1957  3           rbe_log_stats(
0,"RSTPI_GetTopLevelTemplates: Template property not found on backup
                                                                    node");
1958  3           continue;
1959  2       }
1960
1961  2       RnPropertyChar *cp=(RnPropertyChar *)tp;
1962  2       char * tname=(char *)(cp->getValue());
1963
1964  2       //
1965  2       // We should always find a template property.  Skip this item
1966  2       // if we don't.
1967  2       //
1968
1969  2       if(tname[0] == '\0')
1970  3         {
1971  3           rbe_log_stats(
0,"RSTPI_GetTopLevelTemplates: Template property ob backup node has no
                                                                    value");
1972  3           continue;
1973  2         }
1974
1975  2       // Now, let's see if we already have it...
1976
1977  2       bool addIt=TRUE;
1978
1979  2       struct RSTRPC_name_list *lookat_next;
1980  2       struct RSTRPC_name_list *lookat_entry=*templates;
1981
1982  2       while(lookat_entry)
1983  3          {
1984  3            lookat_next=lookat_entry->next;
1985  3            if(0==strcmp(tname,lookat_entry->name))
1986  4            {
1987  4              lookat_next=NULL;
1988  4              addIt=FALSE;  // Fount it!!!
1989  3            }
1990  3            lookat_entry=lookat_next;
1991  2       }
1992
1993  2       //
1994  2       // Add the entry only if it hasn't already been placed on the
                                                                    list
1995  2       //
1996  2       if(addIt)
1997  3          {
1998  3            //
1999  3            // Add entry to the list if not already here...
2000  3            //
2001  3            struct RSTRPC_name_list *entry;
2002
2003  3            entry=(struct RSTRPC_name_list *)malloc(sizeof(
                                                    struct RSTRPC_name_list));
2004
2005  3            if(NULL==entry)
2006  4              {
2007  4                rbe_log_stats(
                            0,"RSTPI_GetTopLevelTemplates: Malloc failure");
2008  4                return EP_RB_RECOVER_MALLOC_FAILURE;
2009  3              }
2010
2011  3            entry->next=*templates;
2012  3            entry->name=esl_strdup(tname);
```

```
2013  3            if(NULL == entry->name)
2014  4              {
2015  4                rbe_log_stats(
                             0,"RSTPI_GetTopLevelTemplates: Malloc failure");
2016  4              return EP_RB_RECOVER_MALLOC_FAILURE;
2017  3              }

2019  3            *templates=entry;

2021  3            (*numberEntries)++;
2022  2          }

2024  1      }


2027  1    return E_SUCCESS;
2028       }
```



```
2031  /*************************************************************
2032   *  Clear Restore Context :
2033   *
2034   *  This function is called to notify the plug-in that its currently
                                                          selected
2035   *  top level object is no longer selected (
                                                  for browsing and marking). The
2036   *  plug-in should perform whatever cleanup and memory deallocation is
2037   *  appropriate.
2038   *
2039   * Returns:
2040   *       E_SUCCESS          on success
2041   *       EP_RB_RECOVER_xxx  on error
2042   *
2043   * Parameters:
2044   *   context          (I)  - Pointer to the restore context
2045   *
2046   *************************************************************/
2047  eerrno_ty RSTPI_ClearRestoreContext( restore_context *context )
2048  1 {
2049  1   //
2050  1   // Get the restore context so we can find the app data
2051  1   //
2052  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData)
2053  1 ;
2054  1     rbe_log_debug(0,"In RSTPI_ClearRestoreContext");

2056  1   if(NULL == rcd)
2057  2     {
2058  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
2059  1     }

2061  1   // Clear locks if they exist
2062  1   if(NULL != rcd->currentBackupNode)
2063  2   {
2064  2     rbe_log_debug(0,"RSTPI_ClearRestoreContext: unlocking %s",
2065  2                   rcd->currentBackupNode->getNameChar());
2066  2     rcd->currentBackupNode->unlockWorkitems(context);
2067  2     rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
2068  1   }
2069  1   rcd->currentBackupNode=NULL;

2071  1           return E_SUCCESS;
2072       }
```

```
2075  /*****************************************************************************
                                                                                *
2076   * Is There Prev Backup For Time
2077   *
2078   * Function Description:
2079   *       Determine if a backup exists prior to the specified time
2080   *
2081   * Parameters:
2082   *       context (I) - Pointer to the restore context
2083   *       thisTime(I) - Time for the query
2084   *       flags   (
                  I) - Backup constraint flags: e.g., full-only/partial-ok
2085   *       isThere (O) - TRUE/FALSE that requested backup does exist
2086   *
2087   * Return Codes:
2088   *       E_SUCCESS                 - operation completed successfully
2089   *       EP_RB_RECOVER_xxx         - when errors occur accessing catalogs
2090   *
2091  *****************************************************************************
                                                                                */


2093  eerrno_ty RSTPI_IsTherePrevBackupForTime( restore_context *context,
2094                                            const time_t      thisTime,
2095                                            RSTRPC_backup_flags_ty  flags,
2096                                            boolean_ty              *isThere )
2097  1 {
2098  1   *isThere=FALSE;

2100  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                        context->appData);



2103  1   WiSetNode *wilist=rcd->currentWisetNode;

2105  1   if(!wilist)
2106  2     {
2107  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
2108  1     }


2111  1   BackupNode *bnode=rcd->currentBackupNode;

2113  1   if(!bnode)
2114  2     {
2115  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
2116  1     }


2119  1   //
2120  1   // Loop over all backup objects in this set looking for one which is
                                                                      previous
2121  1   // to this one.
2122  1   //
2123  1   for(RestoreNode *rn=wilist->getFirstChild();rn;
                                                rn=wilist->getNextChild())
2124  2     {
2125  2       if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                        STATE_COMPLETE))
2126  3           {
2127  3             BackupNode *bn=(BackupNode *)rn;
2128  3             RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                                  PROPERTY_BACKUP_DATE));
2129  3             time_t btime=dp->getValue()->unixTime();
```



```
2131  3             if(thisTime > btime)
2132  4               {
2133  4                 *isThere=TRUE;
2134  4                 return E_SUCCESS;
2135  3               }
2136  2           }
2137  1     }

2139  1     return E_SUCCESS;

2141  }
```

```
2143  /*****************************************************************
                                                                      *
2144   * Is There NextBackup For Time
2145   *
2146   * Function Description:
2147   *      Determine if a backup exists after to the specified time
2148   *
2149   * Parameters:
2150   *      context (I) - Pointer to the restore context
2151   *      thisTime(I) - Time for the query
2152   *      flags   (
                  I) - Backup constraint flags: e.g., full-only/partial-ok
2153   *      isThere (O) - TRUE/FALSE that requested backup does exist
2154   *
2155   * Return Codes:
2156   *      E_SUCCESS               - operation completed successfully
2157   *      EP_RB_RECOVER_xxx       - when errors occur accessing catalogs
2158   *
2159   *****************************************************************
                                                                      */

2161  eerrno_ty RSTPI_IsThereNextBackupForTime( restore_context *context,
2162                                  const time_t            thisTime,
2163                                  RSTRPC_backup_flags_ty  flags,
2164                                  boolean_ty              *isThere )
2165  1 {
2166  1   *isThere=FALSE;

2168  1   struct restoreContextData *rcd=(struct restoreContextData *)(
                                                    context->appData);


2171  1   WiSetNode *wilist=rcd->currentWisetNode;

2173  1   if(!wilist)
2174  2     {
2175  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
2176  1     }


2179  1   BackupNode *bnode=rcd->currentBackupNode;

2181  1   if(!bnode)
2182  2     {
2183  2       return EP_RB_RECOVER_RC_APP_DATA_NULL;
2184  1     }


2187  1   //
2188  1   // Loop over all backup objects in this set looking for one which is
                                                              previous
2189  1   // to this one.
2190  1   //
2191  1   for(RestoreNode *rn=wilist->getFirstChild();rn;
                                          rn=wilist->getNextChild())

2192  2     {
2193  2       if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                    STATE_COMPLETE))
2194  3         {
2195  3           BackupNode *bn=(BackupNode *)rn;
2196  3           RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                                  PROPERTY_BACKUP_DATE));
2197  3           time_t btime=dp->getValue()->unixTime();

2199  3           if(thisTime < btime)
```



```
2200  4             {
2201  4               *isThere=TRUE;
2202  4               return E_SUCCESS;
2203  3             }
2204  2         }
2205  1     }

2207  1     return E_SUCCESS;

2209  }
```

```
2211  /*****************************************************************
2212   * Identify:
2213   *
2214   *  This function is called once,
                                  to identify and validate the plug-in with
2215   *  regard to the operating restore engine.
                                  The version number is checked
2216   *  for compatibility with the restore engine,
                                  and the optional features
2217   *  of the plug-in are specified.
2218   *
2219   * Parameters:
2220   *  pi_defs (
              O) - address of the structure identifying the plugin to the
2221   *              restore engine. Its fields consist of:
2222   *      version  - Version of the plugin header that the library was
                                  built with
2223   *      name     - Name of the backup application (
                                  64 byte buffer address)
2224   *      options  - Bit mask identifying the optional plug-in features
                                  supported
2225   *           The bit definitions for this parameter (
                                  RSTPI_OPTION_MASK...) are
2226   *           defined above.
2227   *      wi_types  - pointer to array of workitem types supported by the
                                  plugin
2228   *      num_types - number of witypes in the witypes array
2229   *
2230   * Returns:
2231   *      E_SUCCESS          on success
2232   *      EP_RB_RECOVER_xxx  on error
2233   *
2234   *****************************************************************/
2235  eerrno_ty RSTPI_Identify( const struct pluginIDdata **pi_defs )
2236 1{
2237 2        static char types[2] = {WI_TYPE_DC_NETWORK,
2238 1                                WI_TYPE_DC_WORKER};
2239 2        static struct pluginIDdata myPiID = {
                                        /* plugin Identification */
2240 2                                        RSTPI_VERSION,
2241 2                                    "Symmetrix Connect Restore
                                                Plug-in",
2242 2                                RSTPI_OPTION_SYMM_RESTORE_ALLOWED,
2243 2                                    types,
2244 2                                    2
2245 1                                };

2247 1        *pi_defs = &myPiID;

2249 1        return E_SUCCESS;
2250  }
```



```
2253  /*****************************************************************
2254   * Finish:
2255   *
2256   *  This function is called to allow the plug-in to clean up its
                                                        internal
2257   *  storage when a restore session is ending.
2258   *
2259   * Parameters:
2260   *  context (I) - Pointer to the restore context
2261   *
2262   * Returns:
2263   *      E_SUCCESS          on success
2264   *      EP_RB_RECOVER_xxx  on error
2265   *
2266   *****************************************************************/
2267  eerrno_ty RSTPI_Finish( restore_context *context )
2268 1{
2269 1   struct restoreContextData *rcd =
2270 1                                   (struct restoreContextData *){
                                                        context->appData);

2272 1   delete rcd->currentWisetListNode;
2273 1   delete rcd->currentWisetNode;
2274 1   delete rcd->currentBackupNode;
2275 1   free (rcd);

2277 1   return E_SUCCESS;
2278  }
```

```
2280  /*****************************************************************
2281   * Does Alternate Exist
2282   *
2283   *  This routine determines if an alternate trailset exists for the
2284   *  given template.
2285   *
2286   * Parameters:
2287   *  context      (I) - Pointer to the restore context
2288   *  templateName (I) - The name of the template to look for
2289   *  exists       (
2290   *                O) - Return flag for whether or not the alternate exists
2291   *
2292   *****************************************************************/
2293  eerrno_ty
2294  RSTPI_DoesAlternateExist( restore_context         *context,
2295                            const template_name_ty template_name_ty templateName,
2296                            boolean_ty              *exists )
2297  1 {
2298  1 /************** TEMPORARY,
                        until we get the real version: ********************/
2299  1         *exists = FALSE;
2300  1         return E_SUCCESS;
2301    }
```